3P17/4P17 COMPUTABILITY 2014-2015 Lecture Handout 5

Computing a function in a Non-deterministic Turing machine

We say a non-deterministic TM N **computes** a function on Σ-strings $f : \Sigma^* \to \Sigma^*$ if on any input $\sigma \in \Sigma^*$, every branch in the computation tree results in an accepting configuration with $f(\sigma)$ in the left-most squares of the tape leaving the others blank.

Note: We may instead have said that every branch that leads to an accepting configuration leaves $f(\sigma)$ on the tape and that there is at least one such branch. This would allow the machine to have infinite branches or branches that lead to rejecting configurations. However, at this point, we know that such a machine will not give any more computational power than a single tape deterministic Turing machine, i.e. any function f which can be computed by such a non-deterministic TM can also be computed by some single-tape deterministic TM. As such, we can insist that all branches lead to accepting states with $f(\sigma)$, happy in the knowledge that we are not missing out on any computational power that might be afforded by the more liberal definition.

Hilbert's tenth problem

Hilbert's tenth problem: *Devise a process according to which it can be determined in a finite number of operations whether a given (multivariate) polynomial $p(x_1, x_2, \ldots, x_r)$ with integer coefficient has an integral root.*

Having an integral root means there is some assignment to the variables $x_1, x_2, \ldots, x_r$ in which every variable takes an integer value in the assignment, and in which gives $p(x_1, x_2, \ldots, x_r) = 0$.

Cast in the framework of languages, a formulation of this problem would be the following: *Let*

$$L = \{p : p \textit{ is a (multivariate) polynomial with integer coefficients and which has an integer root}\}.$$

Is L decidable?

The answer to this question is **no**, L is **not** decidable. It is, of course, recognisable since given any polynomial p, one may progress through all possible integer assignments of the variables in some systematic manner, and if a root exists, it will be found eventually.

Now let $L' \subset L$ be defined as

$$L' = \{p : p \text{ is a univariate polynomial with integer coefficients and which has an integer root}\}.$$

L' is indeed decidable. This comes from the fact that an root of p must lie in the interval $\left[-k\left|\frac{C}{c_1}\right|, k\left|\frac{C}{c_1}\right|\right]$, where k is the number of terms in the polynomial, C is the coefficient with the largest absolute value, and c_1 is the coefficient of the largest order term. One may then search the solution space as follows: $0, 1, -1, 2, -2, \ldots$ possibly until exceeding the bounds. Hence, this approach will certainly terminate in a finite number of steps and it will decide if the candidate polynomial p is in L'.

Crucially, the undecidability of the multivariate case could not have been discovered without a precise definition of "process", and we may define such a process as being a Turing machine.

Binary representation

Let N be a non-negative integer such that $N = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_0 2^0$ where each $a_i \in \{0, 1\}$. Then the string $a_n a_{n-1} \ldots a_0$ is **the binary representation of N in $n+1$ bits**.

The number 37, for example, has binary representation 100101 in 6 bits. It has binary representation 00100101 in 8 bits, but we cannot represent it in less than 6 bits.

For a positive integer N, we write $\langle N \rangle = a_n a_{n-1} \ldots a_0$ for the minimal length binary string representing N. We also write $\langle 0 \rangle = 0$. For example, $\langle 37 \rangle = 100101$.

Note, '37' and '100101' are merely two different ways of representing *exactly* the same thing, that thing being a particular element of the set $\mathbb{N}$. We may, of course, represent it in other ways, such as in ternary or hexadecimal. When we say "binary number" we mean the binary representation of that number.

In a binary string, the leftmost bit is called **the most significant bit** (MSB) and the rightmost bit is called the **the least significant bit** (LSB). In 00100101 the MSB is 0 and the LSB is 1.

To "flip" a bit is to change it to 1 if it is 0 and 0 if it is 1.

Binary arithmetic

To **add 1** to a binary number do the following: Starting at the LSB and going left, flip 1's until reaching the first 0. Flip it into a 1 and stop. If there are no 0's (i.e., the bit was all 1's) then after flipping all the 1's, increase the length of the string by putting a 1 at the front.

e.g $1101 + 1 = 1110$, and $1111 + 1 = 10000$.

To **subtract 1** from a binary number representing a positive integer, starting at the LSB and going left, flip 0's until the first 1. Flip it to 0 and stop.

e.g., $1000 - 1 = 0111$, and $1011 - 1 = 1010$.

Turing machines for arithmetic, composition of Turing machines, multiple input arguments

Let $\Sigma = \{\texttt{0}, \texttt{1}\}$. Suppose we have a Turing machine M_{+1} that computes the function on Σ-strings $f_{+1} : \Sigma^* \to \Sigma^*$, $f_{+1}(\langle N \rangle) = \langle N+1 \rangle$ for $N \in \mathbb{N}_0$, $f(\sigma) = \varepsilon$ for any other $\sigma \in \Sigma^*$. That is, interpreting binary strings as non-negative integers, M_{+1} computes the function $F_{+1} : \mathbb{N}_0 \to \mathbb{N}$, $F_{+1}(N) = N + 1$. If a binary string is empty or is not a minimal representation of some integer, then it returns an empty string.

Give such an M_{+1}, we can create a machine M_{+2} by simulating M_{+1}, copying the output of that simulation back onto the input, and simulating M_{+1} again. We can create M_{+3}, M_{+4} and so on in a similar way.

Suppose we have a Turing machine M_{-1} that computes the function on Σ-strings $f_{-1} : \Sigma^* \to \Sigma^*$, $f_{-1}(\langle N \rangle) = \langle N-1 \rangle$ for $N \in \mathbb{N}$, $f(\sigma) = \varepsilon$ for any other $\sigma \in \Sigma^*$. That is, interpreting binary strings as non-negative integers, M_{-1} computes the function $F_{-1} : \mathbb{N} \to \mathbb{N}_0$, $F_{-1}(N) = N - 1$. Note, M_{-1} does not deal with negative numbers, but any $\sigma \in \{\texttt{0}\}^*$ is a possible input string, so it must return some value. We choose ε, which it also returns for binary strings which are not minimal representations.

We can then create M_{-2}, M_{-3}, etc using M_{-1}.

Turing machines that are simulated in the definition of other Turing machines are sometimes called "modules", "subroutines" or "components". A Turing machine M using another Turing machine M' as a subroutine can be thought of as M putting input onto M''s input tape, letting it run, then taking output from M''s output tape, or if M' is being used to recognise/decide to just make a decision based on the state the

M' ended up in (if it halted). In reality, of course, M is the only Turing machine, and the behaviour of M' is encoded into M's 7-tuple definition.

We do not want to define a Turing machine for every number we might want to add or subtract, so we will need to provide two or more arguments.

Computing functions with multiple arguments: We can give a Turing machine a multi-argument input by using delimiters. e.g., $\Sigma = \{\mathtt{0}, \mathtt{1}, *\}$ and input might be $\mathtt{10010} * \mathtt{0} * \mathtt{110}$. Since any member of Σ^* can be given as input, the machine needs to check that the correct form is given, so if an input like $\mathtt{1010} * *$ is invalid the machine should catch it.

Assume we have a machine $M_{\text{check}\langle A\rangle * \langle B\rangle}$ whose input alphabet is $\Sigma = \{\mathtt{0}, \mathtt{1}, *\}$ and which accepts if the input has form $\langle A\rangle * \langle B\rangle$ for some non-negative integers $A, B \in \mathbb{N}_0$, otherwise it rejects. That is $M_{\text{check}\langle A\rangle * \langle B\rangle}$ decides the language $L = \{\langle A\rangle * \langle B\rangle : A, B \in \mathbb{N}_0\} \subset \Sigma^*$.

We can create a two-tape Turing machine M_{A+B} to add non-negative integers with the result on tape 2. We call the following format of Turin machine description an **implementation-level description** . Any Turing machines used in an implementation-level description must themselves be known to exist.

M_{A+B}
Let $\Sigma_1 = \{\mathtt{0}, \mathtt{1}, *\}$. M_{A+B} computes $f_{A+B} : \Sigma_1^* \to \Sigma_1^*$, $f_{A+B}(\langle A\rangle * \langle B\rangle) = \langle A + B\rangle$ for $A, B \in \mathbb{N}_0$, and $f_{A+B}(\sigma) = \varepsilon$ for any $\sigma \in \Sigma_1^*$ not of that form. Input is on tape 1, output is on tape 2.

(1) Simulate $M_{\text{check}\langle A\rangle * \langle B\rangle}$ on the input. If it rejects then halt with accept with ε on tape 2. Otherwise goto (2).

(2) Copy $\langle B\rangle$ onto tape 2, delete the delimiter from tape 1, leaving just $\langle A\rangle$.

(3) If $\langle A\rangle = \mathtt{0}$, halt with accept. Otherwise simulate M_{-1} on tape 1 and M_{+1} on tape 2.

(4) go to (3).